Mise en œuvre d'un SE d'ordre 0+

Weixuan XIAO (GI04)

Yan LIU (GI04)

Context

1 Introduction

Notre projet a pour but de détecter du langage de programmation. Parce qu'il y a trop de langages de programmation, on a pris des langages classiques et connus pour réduire la complexité et augmenter la précision. Mais il y a aussi une possibilité de faire une extension, on le parlera dans la génération de règles. Les langages que l'on a pris en compte sont C, C++, Java, JavaScript, Python et Lisp.

Ce projet est inspiré par une thèse sur statistique : Programming-Language-Identification. On voudrait la réaliser sous Lisp dans un forme de Système Expert d'ordre 0+, qui corresponde aux consignes de TP2 et TP3 d'IA01.

Des outils simples d'identification du code source existent déjà. Divers outils de mise en évidence de la syntaxe, tels que Google Code Prettify, met automatiquement en évidence la syntaxe en fonction du code. Cependant, ces outils n'identifient pas réellement les langues; au lieu de cela, ils utilisent des heuristiques qui feront que la mise en évidence fonctionne bien. Dans le cas de Google Code Prettify, les grammaires larges (telles que C-like, Bash-like et Xml-like) sont préprogrammées. Ces grammaires sont ensuite utilisées pour analyser le code et la grammaire la mieux adaptée est utilisée pour la mise en évidence. Clairement, les langues qui partagent une grammaire ne peuvent pas être distinguées.

En même temps, sur GitHub, on utilise le nom d'extension comme « .c », « .py » pour détecter le langage. Ce n'est pas précis. Par exemple si on écrit des codes C dans un fichier avec un nom d'extension « .cpp », GitHub le traitera comme un fichier C++.

En combinaison les deux idées, on peut rédiger un système expert pour détecter les langages de programmations. Pour détecter le langage d'une source de code, il faut savoir quelques caractéristiques de la source selon la thèse. Par exemple : LastCharacter, FirstWord, Punctualistion, Bracket, Operator, KeyWord, etc. Il y a encore des autres caractéristiques comme « lineComments.txt », etc. Mais le traitement de texte est très complexe, on le laisse comme un point d'amélioration au futur. Pour résoudre les problèmes dans le monde réel, on a décidé faire un système expert qui peut détecter le fichier du code puis entrer dans le système expert.

Pour faire ce moteur avoir une grande crédibilité et faciliter l'utilisation, le traitement du fichier et les règles pour la base doivent avoir des preuves. Donc, il nous a pris grand effort dans cette partie du traitement du fichier de la source code.

2 Résoudre par un SE

Avoir beaucoup de caractéristiques pout juger et beaucoup de relation entre ces caractéristiques, sous lisp il faut utiliser le système expert qui peut tirer des conclusions à partir de relations complexes et automatiser une tâche routinière. Et pour répondre d'exigence de détecter le type du langage, c'est bien le moteur d'ordre 0+ avec l'algorithme chaînage -avant.

Pourquoi l'ordre 0+ pour le moteur ?

Parce que chaque caractéristique doit bien être spécifiques, ça va dire le fait est un couple (attribut, valeur) ou une triple (objet-attribut-valeur) : AV ou OAV.

Voici un exemplaire de règle pris en compte de notre moteur d'inférence. Source est un objet qui stocke les caractéristiques de code source à détecter. Les structures seront présentées dans les chapitres suivants.

```
(R1
   (
      (equal (?punctuation source) (intern ";"))
   )
   (
      (equal (punctuation-probability c) T)
   )
)
```

On a utilisé les règles origines de la base de données de la thèse, dont le format est une paire « clé-valeur ». La valeur est le Score et la clé est le caractère ou le mot dans la source. Nous avons traité les données dedans et nous les transformés. On prendra le mot ou le caractère plus fréquent, si il a une bonne score dans une langage, on pense qu'il y a une possibilité de devenir ce langage.

e 9080

i 71

m 943

q 25

u 28

y 1108

} 18309

En fait, dans la thèse origine, c'est un système de calcul de probabilité avec les caractéristiques données. Mais notre système doit être en ordre 0+. Ainsi on a fait quelques codes pour générer les règles en ordre 0+. Avant le démarrage de système, on peut ajouter des critères, on peut régénérer les règles en appelant le code « generate-regles.lisp ». C'est également pour la possibilité d'extension (On peut dynamiquement ajouter des langages et des critères dans notre système pour la possibilité de détection de plus de langage et de détection plus précise).

3 Prétraitement du fichier

Notre système est fonctionné par analyser le fichier du code source. Donc, la première partie c'est le traitement du fichier.

3.1 Traitement de fichier

Pour détecter une source du code, on ne peut pas calculer les caractéristiques manuelles puis entrer la base des faits. Il faut traiter la source du code, calculer les caractéristiques utili pour gérer une base de faits automatique sous un format unique. (Présenter dans la partie suivante 3.1.1 Format d'un fait)

Pour détecter le type du langage : Il faut saisir ses LastCharacter, FirstWord, Punctualistion, Bracket, Operator, KeyWord. Dans la thèse origine, ils ont pris tous les caractères et leur ont donnés des Scores. Pour la simplification, on a décidé de prendre que les mots ou les caractères, pour chaque critère, dont la fréquence d'apparaitre est la plus élevée.

Exemple pour récupérer les « LastCharacter » du code de source :

```
(load "parsing-module-symbol.lisp")
(defun get-last-character (source)
(let ((characters '()) (f (open source :direction :input)) (num-lines 0))
(loop
  (let* ((line (read-line f NIL)) (trimed-line (string-trim " " line)))
        (if line
              (if (> (length trimed-line) 0)
            (let ((ch (intern (subseq trimed-line (- (length trimed-line) 1) (length
trimed-line)))))
              (setf num-lines (+ num-lines 1))
              (if (assoc ch characters)
              (setf (cdr (assoc ch characters)) (+ (cdr (assoc ch characters)) 1))
              (push (cons ch 1) characters))))
            (return))))
      characters))
```

Exemple pour récupérer le LastCharacter qui est en fréquence la plus haute :

```
(defun get-highest (pairs)
```

```
(let* ((highest-pairs '()) (sorted-pairs (sort pairs #'sort-dsec-helper))
(highest (car sorted-pairs)))
    (if highest ;; If highest exists
        (let ((rest-pairs (cdr pairs)))
            (push highest highest-pairs)
            (loop
                (if rest-pairs
                    (if (equal (cdr (car rest-pairs)) (cdr highest))
                        (progn
                            (push (car rest-pairs) highest-pairs)
                            (setf rest-pairs (cdr rest-pairs))
                        )
                        (return ))
                    (return )))))
    highest-pairs))
```

Difficultés rencontrées : Pour récupérer les keywords et opérateurs, le traitement du string dans lisp est plus difficile que l'autre langages. Cette partie nous a pris plus du temps.

3.2 Représentation Object

Puisque le processus du traitement du fichier, il doit utiliser le principe de la Représentation Object. D'après la construction Object on peut gérer la base de fait, qui récupérer les paramètres viennent du traitement.

Exemple d'utilisation :

```
(defclass $probability ()
   (
      (lastCharacter-probability :accessor ?lastCharacter-probability :initarg :lastcharacter-probability :initform NIL)
      (firstword-probability :accessor ?firstword-probability :initarg :firstword-probability :initform NIL)
      (punctuation-probability :accessor ?punctuation-probability :initarg :punctualition-probability :initform NIL)
      (brackets-probability :accessor ?brackets-probability :initarg :brackets-probability :initform NIL)
      (operators-probability :accessor ?operators-probability :initarg :operators-probability :initform NIL)
      (keywords-probability :accessor ?keywords-probability :initarg :keywords-probability :initform NIL)
   )
)
```

```
(defparameter languages '())
(defparameter c (make-instance '$probability))
(push c languages)
(defparameter cplusplus (make-instance '$probability))
(push cplusplus languages)
(defparameter python (make-instance '$probability))
(push python languages)
(defparameter java (make-instance '$probability))
(push java languages)
(defparameter javascript (make-instance '$probability))
(push javascript languages)
(defparameter lisp (make-instance '$probability))
(push lisp languages)
```

4 Mise en place du moteur d'inférence

4.1 Quelques définitions importante

4.1.1 Présentation d'un fait

Le fait est géré automatiquement après le traitement du fichier, le format pour représenter un fait quelconque est le suivant :

((lastCharacter . t) (firstword .)) (punction . ;) (brackets .)) (operators . ;) (keywords . assert))

4.1.2 Présentation d'une règle

Nous avons choisi le format de règle suivant (Ordre 0+) :

(SI (premisse1 premisse2 ...) ALORS (conclusion1 conclusion2 ...))

Car notre base de fait vient du le traitement du fichier est généré sous la forme présentée dans la partie avant (3.1.1).

Donc pour à la fin pouvoir juger le type du langage, il faut avoir les règles de trois niveaux :

```
(R1
    (
        (equal (?punctuation source) (intern ";"))
    )
    (
        (equal (punctuation-probability c) T)
    )
)
```

```
(R395
    (
        (equal (?LASTCHARACTER-PROBABILITY C) NIL)
        (equal (?FIRSTWORD-PROBABILITY C) T)
        (equal (?PUNCTUATION-PROBABILITY C) NIL)
        (equal (?BRACKETS-PROBABILITY C) T)
        (equal (?OPERATORS-PROBABILITY C) NIL)
        (equal (?KEYWORDS-PROBABILITY C) NIL)
    )
    (
        (equal (C *result*) 2)
    )
)
```

```
(R779
    (
        (>= (?python *result*) 4)
    )
    (
        (print "Le code est probablement python
")
    )
)
```

Code Lisp pour Générer des règles automatique :

```
(defun pickup-features (features-list min-score)
    (let ((features-filted '()) (item (car features-list)) (features (cdr features-list)))
        ; For a sorted list, we can stop when the
        (loop
            (if features
                (if (>= (cdar features) min-score)
                    (push (car features) features-filted)
                    (return)
                )
                (return)
            )
            (setq features (cdr features))
        )
        features-filted
    )
)
(setq count 0)
(setq regles-origin '())
(dolist (score *scores*)
    (dolist (part (cdr score))
        (if (member (intern (car part)) (list (intern "keywords") (intern "operators")))
            (let ((features (pickup-features (cdr part) 5000)))
                ;(setf count (+ count (length features)))
                (dolist (feature features)
                    (setf count (+ count 1))
                    (format T "(R~A
    (
        (equal (?~A source) (intern \"~A\"))
    )
    (
        (equal (~A-probability ~A) T)
    )
)~%" count (car part) (car feature) (car part) (car score))
                )
            )
            (let ((features (pickup-features (cdr part) 1000)))
                ;(setf count (+ count (length features)))
                (dolist (feature features)
```

```
                        (setf count (+ count 1))
                        (format T "(R~A
    (
        (equal (?~A source) (intern \"~A\"))
    )
    (
        (equal (~A-probability ~A) T)
    )
)~%" count (car part) (car feature) (car part) (car score))
                    )
                )
            )
        )
)
```

4.1.1 Notion de règle candidate

Une règle candidate est une règle dont les prémisses sont vérifiées compte tenu de l'état actuel de la base de faits. Il s'agit donc d'une règle qui fournit à coup sûr une information nouvelle si on choisit de la déclencher. La fonction de service regles-candidates suivante retourne la liste des identifiants des règles candidates en fonction des informations contenues actuellement dans la base de faits (donc NIL si aucune règle n'est candidate). L'algorithme qui régit le fonctionnement de cette fonction est le suivant

```
(defun get-regles-possible (regles)
    (let ((regles-valable '()))
        (dolist (regle regles)
            (if (valid-et (premisses regle))
                (if (not (member regle *applied-regles*))
                    (progn
                        ;(format T "Regle ~A Result: ~A  ~A ~%" (nom regle) (valid-et (premisses regle)) (conclusions regle))
                        (push regle regles-valable)
                    )
                )
            )
        )
        regles-valable
    )
)
```

La fonction de service valid-premisse suivante permet de vérifier si une prémisse d'une règle est vérifiée en fonction des informations contenues dans la base de faits (qui doit obligatoirement contenir un fait permettant de décider). Elle prend en argument une prémisse ainsi que le fait de la base de faits qui permet de conclure (Il a été vérifié en dehors de la fonction que ce fait existe bien). Elle retourne alors T si la prémisse est vérifiée et NIL sinon :

La fonction de service valid-et suivante permet de vérifier une liste de premisse, puis vérifier cahque premisse comme valid-premisse :

```
(defun valid-et (premisses)
    (let ((result T))
        (dolist (premisse premisses)
            (if (not (valid-premisse premisse))
                (setf result NIL)
            )
        )
        result
    )
)

(defun valid-premisse (premisse)
    (funcall (car premisse) (eval (cadr premisse)) (eval (caddr premisse)))
)
```

La fonction de service apply-conclusion:

```
(defun apply-conclusion (conclusion)
    (if (equal (car conclusion) 'equal)
        (progn
            (setf (slot-value (eval (cadadr conclusion)) (caadr conclusion)) (eval (caddr conclusion)))
            NIL
        )
        (progn
            (eval conclusion)
            T ; Resultat trouve
        )
    )
)
```

4.2 Principe du chaînage -avant en largeur d'abord

A chaque cycle du moteur d'inférence, toutes les règles candidates, identifiées en fonction du contenu actuel de la base de faits, sont déclenchées.

Le code LISP correspondant est le suivant :

```
 ;; Breadth first search, O(n)
(defun parcour-regles (regles applied-regles depth)
    (let ((conclusion-a-apply '()) (is-ok NIL))
        (dolist (regle regles)
            (if (valid-et (premisses regle))
                (if (not (member regle applied-regles))
                    (progn
                        (push (conclusions regle) conclusion-a-apply)
                        (push regle applied-regles)))))
        (if (= (list-length conclusion-a-apply) 0)
            (format T "Inference over~%")
            (progn
                (dolist (conclusions conclusion-a-apply)
                    (dolist (conclusion conclusions)
                        (if (apply-conclusion conclusion)
                            (setf is-ok T))))
                ;; Enter next
                (let ((next-ok (parcour-regles regles applied-regles (+
depth 1))))
                    (or next-ok is-ok) ))))
```

Commentaire :

Chaque fois on prendra toutes les règles qui sont valables dans une itération. S'il en a, on les applique et on continue les itérations suivantes. Sinon, ça veut dire c'est terminé.

On a plusieurs possibilités donc on doit parcourir toutes les règles, on ne s'arrête pas quand une conclusion finale se produit.

Donc on détecte si les conclusions sont validées avec « is-ok » et la retourne.

4.3 Principe du chaînage -avant en profondeur d'abord

A chaque cycle du moteur d'inférence, seule la première des règles candidates est déclenchée (Il ne s'agit que d'un choix stratégique parmi d'autres possibles).

Le code LISP correspondant est le suivant :

```
;; Depth first search
(defun depth-regles (regles)
    (let ((regles-valable (get-regles-possible regles)) (is-ok NIL))
        ; Get valable
        ; Pick up one, valide it, enter next
        (dolist (regle regles-valable)
            (dolist (conclusion (conclusions regle))
                (push regle *applied-regles*)
                (if (apply-conclusion conclusion)
                    (progn
                        (setf is-ok T)
                    )
                )
                (let ((next-ok (depth-regles regles)))
                    (if next-ok
                        (progn
                            (setf is-ok next-ok)
                            (return )))
                )
            ))
        is-ok))
```

Commentaire :

Chaque fois on prendra une des règles qui est valable dans une itération. S'il en a, on l'applique et on continue les itérations suivantes. Sinon, ça veut dire ce n'est pas suffisant pour conclure. Donc on retourne et prend les autres conditions.

On a plusieurs possibilités donc on doit parcourir toutes les règles, on ne s'arrête pas quand une conclusion finale se produit.

Donc on détecte si les conclusions sont validées avec « is-ok » et la retourne.

5 Scenarios Test

5. 1 Scenarios avec une base de fait de 4 caractéristiques

Cette fois on sélect 4 caractéristiques pour gérer les bases de faits du départ :

On fait tourner le moteur d'inférence : On peut voir que pour les faits du départ, le résultat par en profondeur d'abord et en largeur d'abord.

```
PS C:\Develop\IA01\assignment-ia01\tp\TP3> clisp .\run-depth-first.lisp
Le premier mot : (      for . 31)
Le crochet : () . 196) (( . 196)
Le dernier character: (: . 136)
La ponctuation: (; . 2201)
----------- Marchant -----------
Le code est probablement python
Le code est probablement javascript
Le code est probablement cplusplus
Le code est probablement c
Resultat trouve
PS C:\Develop\IA01\assignment-ia01\tp\TP3> clisp .\run-breadth-first.lisp
Le premier mot : (      for . 31)
Le crochet : () . 196) (( . 196)
Le dernier character: (: . 136)
La ponctuation: (; . 2201)
----------- Marchant -----------
Le code est probablement c
Le code est probablement cplusplus
Le code est probablement javascript
Le code est probablement python
Inference over
Resultat trouve
```

Commentaire :

Dans le cas que 4 caractéristiques entrent dans les faits du départ, le système peut donner une démonstration pour le but mais le résultat n'est pas précis. Il nous réponse plusieurs possibilités. C'est parce que on juste choisi 4 critère pour détecter et seulement avec la fréquence.

5. 2 Scenarios avec une base de fait de 6 caractéristiques

On sélect 6 caractéristiques pour gérer les bases de faits du départ, on ajouter les deux autres caractéristiques : (La même fichier détecter)

```
PS C:\Develop\IA01\assignment-ia01\tp\tp3> clisp .\run-depth-first.lisp
Le premier mot : (      for . 31)
Le crochet : () . 196) (( . 196)
Le dernier character: (: . 136)
La ponctuation: (; . 2201)
Le mot cle: (lang . 123)
L'operateur: (_ . 222)
---------- Marchant ----------
Le code est probablement python
Le code est probablement javascript
Le code est probablement cplusplus
Resultat trouve
PS C:\Develop\IA01\assignment-ia01\tp\tp3> clisp .\run-breadth-first.lisp
Le premier mot : (      for . 31)
Le crochet : () . 196) (( . 196)
Le dernier character: (: . 136)
La ponctuation: (; . 2201)
Le mot cle: (lang . 123)
L'operateur: (_ . 222)
---------- Marchant ----------
Le code est probablement cplusplus
Le code est probablement javascript
Le code est probablement python
Inference over
Resultat trouve
```

Commentaire :

Avec plus de caractéristiques, c'est sûr le résultat va être plus précis, il nous donne moins de possibilités.

Les critiques :

Notre système d'expert a encore des points insuffisants qui peuvent être amélioré. Tout d'abord, comment choisir la valeur de chaque caractéristique. Ici, on a utilisé la fréquence la plus haute comme le critère de sélection, mais pour avoir une détection plus précise et fiable, chaque symbole sélecté pour la caractéristique doit avoir une valeur du poids. Pas seulement la fréquence.

6 Scénarios d'utilisation

Étape1 :

Upload le fichier du code dans le répertoire du projet.

Étape2 :

Modifier le paramètre du nomdans fichier ***file.lisp*** à le nom du fichier détecter :

Eemple :

```
1   (setq filename "identifytraits.py")
```

Étape3 :

Exécuter le moteur sous terminal :

```
PS C:\Develop\IA01\assignment-ia01\tp\tp3> clisp .\run-depth-first.lisp
```

7 Comparaison ces deux moteurs

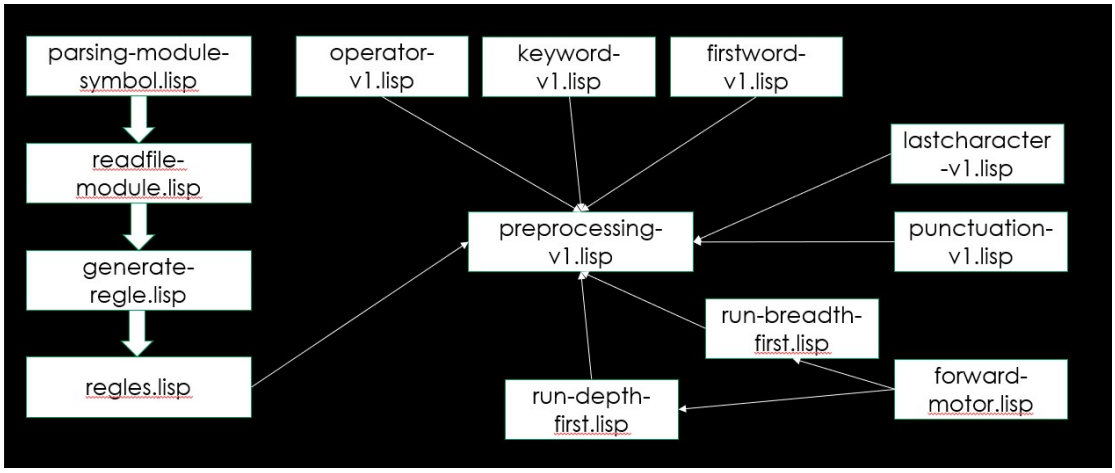
Dans notre système expert, on utilise le moteur de la chaînage -avant en profondeur d'abord : A chaque cycle du moteur d'inférence, seule la première des règles candidates est déclenchée (Il ne s'agit que d'un choix stratégique parmi d'autres possibles).

Et largeur d'abord : A chaque cycle du moteur d'inférence, toutes les règles candidates, identifiées en fonction du contenu actuel de la base de faits, sont déclenchées.

Pour détecter l largeur d'abord, il prendre tous les faits et tous ses prémisses en même temps donc la vitesse est plus rapide.

Mais pour la profondeur d'abord, elle peut trouver la première règle qui correspondante du résultat avec minimum des faits. Dans ce cas-là, il plus lentement.

8 Architecture de système



9 Référence

https://arxiv.org/pdf/1106.4064.pdf